CS 1331: Introduction to Object Oriented Programming
Section A, Spring 2017

Homework 7

Due: Thursday, March 16th, at 8:00 PM

Stefan Young

1 Introduction

For this assignment, we will see some applications of polymorphism in one of the most common classes of algorithms in Computer Science: Sorting. Do not be afraid just because we're talking algorithms - you are more than prepared to take on this assignment.

Today, we enter the fast-paced world of startup culture in Silicon valley. I have a startup idea that brings everything together: Big Data, Machine Learning, Computation Biology, Quantum Mechanics - all on the Cloud. The only problem is, I don't quite have enough employees to realize my dream.

That's where you come in.

2 Problem Description

This week, I'll be your customer and you, on your way to being an OOP maestro from studying under the tutelage of John Stasko, will build an employee management system for me. These won't be any normal employees however, I'm going to need the cream of the crop.

In Silicon Valley terms, I'm going to need some TenXers (read: Ten-Ex-Ers): engineers, managers, ceos, cfos, marketers, and everyone in between. What makes these people TenXers, however, is that they can accomplish 10x the work of a normal employee. That's what I need.

You will be creating classes to model the different types of employees and implementing sorting algorithms to make my hiring decisions easier.

3 Learning Objectives

1. Gain experience in implementing basic sorting operations.
2. Implement a hierarchy of classes to get practice with Polymorphism.
3. Leverage Polymorphism for applications in sorting.
4. Understand the implication of runtime complexity on your program as your input size scales. Otherwise known as Big-Oh.
5. Gain experience using APIs to take advantage of the work done by other developers.
6. Practice testing your code

4 Before We Begin

Like all homework descriptions, it is **very important you read and understand this entire document**. This homework will focus on details.

This homework may be a bit daunting. You should **start early** and come to the TA lab with well thoughtout questions.

You **MUST** be sure to name all your classes and methods **EXACTLY** as described in the description. Let me repeat, you **MUST** follow our naming conventions **EXACTLY**.

5 Provided

1. `SiliconValley.class` This class defines a handful of helpful methods that create random instances of the classes you will implement so you can test your code. **To find out what methods are supplied**, open `SiliconValley.html` in your favorite browser. This file can be found in the **javadoc** folder. Look at the hints in section `6.3` for more description.

2. `SortingRuntime.class` This class file is a simple graphics application that you will run in section `7` to visualize the comparative runtimes of the sorting algorithms you implemented.

3. `Sorting.java` This file defines the method signatures for the sorting algorithms you will implement. **DO NOT CHANGE THE CLASS NAME OR PUBLIC METHOD HEADERS**. The private helper methods you see there are used in most recursive implementations of `MergeSort`. You may use them if you choose to do so.

6 Solution Description Part 1 - Silicon Valley

In this first part, you will have to implement four different classes that will model the different TenXers needed for my team: `TenXer.java, SoftwareEngineer.java, Ceo.java`, and `Cfo.java` .
Each of these should be in their own solution file.

6.1 TenXer.java

This class is the parent of all our different types of TenXers. For my startup, I'm only looking for SoftwareEngineers, Ceos, and Cfos. As such, you should not be able to instantiate an instance of TenXer.
If you want your implementation to work with the provided programs, you have to define your constructors in a particular way that agrees with the assumed formal parameter listing of the helper methods in `SiliconValley.class` and `SortingRuntime.class`
The formal parameters for `TenXer`'s constructor should be:

```
public TenXer(String name, int salary, int yearsExperience
```

1. A `TenXer` should have the following attributes:

   (a) A String representing their name

   (b) An int representing their annual salary

(c) An int representing their years of experience

2. `TenXer` should override `toString()` to return a String representation of all its attributes as well as the class name.

3. `TenXer` should have a properly overriden `equals()` method.

4. `TenXer` The natural ordering of TenXer instances should be by yearsExperience (high to low), salary (low to high: I'm on a budget), and finally their name (alphabetically). Look to the *hints* section to better understand the proper implementation.

6.2 Subclasses of TenXer

Now that you've implemented `TenXer.java`, we're going to have you implement three subclasses which extend `TenXer`: `SoftwareEngineer.java`, `Ceo.java`, and `Cfo.java`.
Again, **so that your implementation works with the provided code**, you must ensure your constructors are as follows. The first three parameters for each subclass should be the same as in `TenXer`. Next should follow the attributes for each subclass. `Ceo` and `Cfo` have only one additional instance variable, so their ordering should be intuitive. For `SoftwareEngineer`, however, `githubStars` should **precede** `knowsJava`.
In addition to the normal work done to extend a class, we will have you **override compareTo() in each class**. Your `compareTo()` methods should look like the sample below.

```
@Override
public int compareTo(TenXer other) {
   if (other instanceof SubclassName) {
      // youre code here
   } else {
      return getClass().getCanonicalName().compareTo(
         other.getClass().getCanonicalName());
   }
}
```

You may be wondering what's going on with `getClass()` and `getCanonicalName()`. In short, this will ensure that we group the subclasses by type. Given an array of `TenXer`'s, we'd like it to be the case that all the `Ceo` are grouped together, all the `Cfo` are grouped together, and all the `SoftwareEngineers` are grouped together. `getCanonicalName()` returns `String` representation of the class name.
Your job for extending the implementation of `compareTo()` is to fill in the `if` block in the above schema for each of the subclasses according to the additional attributes we define below.
Additionally, each subclass must **override** `equals()`, `toString()`, and provide getters for **all** instance variables.

6.2.1 SoftwareEngineer.java

This class represent the workhorse of most modern startups. With the ubiquity of tech in the bay, we're going to need strong engineers with lots of experience.

1. A `SoftwareEngineer` has an 'is-a' relationship with TenXer
2. A `SoftwareEngineer` should have all the attributes of a TenXer, as well as two others
   (a) An int githubStars which measures how prolific the engineer is on github open source projects.
   (b) A boolean knowsJava that denotes whether or not the engineer knows Java.
3. Finally, we're going to override compareTo. As noted in section `6.3` (HINTS), the parameter of this method is of type `TenXer`. That means we might have to compare to instances of `Ceo` or `Cfo`. If that's the case, we're just going to use the `getClass().getCanonicalName()` shortcut that we described at the beginning of this section.

   However, if you are passed in an instance of `SoftwareEngineer`, you should extend the natural ordering of `SoftwareEngineers` to order by githubStars (more is better) and whether or not they know Java (I'd prefer employees who know Java). If the two engineers are tied in these two regards, just return `TenXer`'s `compareTo`.

6.2.2 Ceo.java

This class represents the face of the startup. Your Ceo will be out at VCs raising money, managing the hiring of new employees, and get the company name out there, promoting the company's vision.

1. A `Ceo` has an 'is-a' relationship with TenXer.
2. A `Ceo` should have one new attribute for us to discern which of these `Ceos` is the best for our unicorn startup
   (a) An int numStartups that we'll use to guage their experience in Silicon Valley
3. Again, we'll override `compareTo()`. Like before, if we're not comparing two `Ceos`, we're just going to use the `getClass().getCanonicalName()` shortcut that we described at the beginning of this section.

   If the `TenXer` parameter is a `Ceo`, then we'll want to order them first by their number of startups (descending), then by `TenXer`'s natural ordering if the two `Ceos` have started the same number of companies.

6.2.3 Cfo.java

Every company needs money, and money needs to be managed, or else you end up with a whole lot sit-to-stand tables, ergonomic keyboards, and microkitchens, with little money left over for servers, advertising, and actual product dev.

1. A `Cfo` has an 'is-a' relationship with TenXer.
2. A `Cfo` has one more relevant attribute than TenXer

(a) An int moneySkills that represents how good they are with money (higher is better)

3. For the final time, we'll override `compareTo()`. Again, if we aren't comparing two `Cfo`s, we're just going to use the `getClass().getCanonicalName()` shortcut that we described at the beginning of this section.

   `Cfo`'s, when compared to one another, have a natural ordering according to their moneySkills (larger is better) and, in the case of a tie, the same natural ordering as `TenXer`.

6.3 Hints

1. All classes that you write should have **getter methods** for each piece of instance data.

2. Since we've added new instance variables, all classes you implement should override `toString()` and `equals()`. That being said, make sure you're taking advantage of the work done in each classes' parent in these methods.

3. Whenever possible, use **super** calls in constructors and methods to minimize the amount of code you have to write. We will be looking for proper use of super in grading.

4. We will be implementing `Comparable` in this assignment and take advantage of method overriding to tweak the definition of `CompareTo()` for each type of TenXer we might hire. You should be comfortable with what Comparable means, what contract you are signing when you implement it, and what the returned value of the method means in terms of the relative ordering of the two instances. In this homework, we will not be using the raw type of `Comparable`.

5. From the previous bullet, you might be asking what a *raw type* actually is. In previous homeworks and class, you will likely have seen a class, `MyComparableObject`, for example, implement `Comparable` as such:

```
public class MyComparableObject implements Comparable {
      public int compareTo(Object other) {}
}
```

   Here we used the *raw type* because we never specified what exactly we expect to compare to, so Java, every cautious as it is, forces us to accept all objects and deal with the casting ourselves.

   In practice, however, most implementations of `Comparable` define the *type* they expect. We'll use the following header for `TenXer` on this homework.

```
public class TenXer implements Comparable<TenXer> {
      public int compareTo(TenXer other) { }
}
```

   Of course, you will actually implement the `compareTo()`, and the rest of `TenXer`, but we describe the class header here for clarity's sake. With all subclasses of `TenXer`, be sure you are *overriding* `compareTo()`, **not** *overloading*.

6. Finally, we have provided to you two class files - `SiliconValley.class` and `SortingRuntime.cl`
   `SiliconValley` defines a handful of methods that you might find helpful in testing your code - predominantly creating randomized instances of the different classes you will implement. This might be **useful for testing** your implementations of `compareTo()` and your various sorts.

   You can find the definition of the methods in `SiliconValley` in the `javadoc` folder. `SiliconValley.html` contains all relevant information (the other files are automatically generated by `javadoc`. Don't worry about `SortingRuntime` for now, it will come into play in a later section.

7 Solution Description Part 2 - Sorting

Now that you've designed a model for the TenXer employees looking for jobs in the bay area, we need to implement a system for sorting them. We're going to have you implement two common sorting algorithms, InsertionSort and MergeSort.

You will implement both sorts in a file called `Sorting.java`.

A couple notes:

1. We want our sorts to be as general as possible. We could make these sorts accept an array of `TenXer`s, but then we would be unnecessarily restricted to sort these four (really three) classes. A better design decision would be to sort a polymorphic array of `Comparable` objects. Because any object that implements the `Comparable` interface **must** implement a `compareTo()` method, we are guaranteed to have an implementation of it at runtime, whatever it may be.

2. Remember, `compareTo` returns a negative number, not necessarily -1, if the caller object is less than the parameter, 0 if they are equal, and a positive number, not necessarily 1, if the caller is greater.

   Less than, greater than, and equal to are all defined by the *natural ordering* of the `Comparable` objects - in this case `TenXer` and its subclasses.

   Be sure to consider the implication of these return types when ordering by ascending or descending values.

3. When you compile `Sorting.java`, you may get a warning for unchecked types. You can ignore this - it has to do with our use of `Comparable[]` without explicit type checking. If you want to get rid of it, you can put the `@SuppressedWarnings("unchecked")` annotation immediately above the class header.

4. We have provided pseudo code for each of the two algorithms. You do not need to follow them to the letter - they are included only for you benefit as a reference.

7.1 InsertionSort

Insertion sort operates under the basic idea that if the first n many elements of an array are already sorted, we just have *insert* the $n + 1^{th}$ element into the sorted array by shifting all elements in the presorted array up by one.

To apply this rule, we start with the second element in the array (if there are fewer than two elements, the array is sorted). At this second element, we know that all the elements before it (the first one) are already sorted, so all we have to do is place the second element in the right position relative to those before it. This pattern continues iteratively until the last element of the array is inserted into the right place.

Function *InsertionSort(array A)*
Data: Array of **Comparable** objects
Result: The original array now sorted by the **compareTo**
for $i = 2$ ***to*** $array.length$ **do**
$key = A[i]$
$j = i - 1$
while $j > 0$ *and* $A[j] > key$ **do**
$A[j + 1] = A[j]$
$j - -$
end
$A[j + 1] = key$
end

Algorithm 1: InsertionSort psuedo code

7.2 MergeSort

MergeSort is one of the most powerful and stable sorting algorithms used. MergeSort is a *recursive* algorithm - it splits the array in half at each recursive call until the array has been split into n many one-element arrays. Once we have one-element arrays, we know we can sort them - they are inherently sorted. From this point, we can *merge* the smaller arrays into bigger ones and reorder as needed. By the end of the algorithm, we will have merged many pairs of sorted sub-arrays into our original array, which is now sorted.

Function *MergeSort(array A, array $temp$ int $left$, int $right$)*
Data: Arrays of **Comparable** objects A and $temp$, left endpoint $left$ and right endpoint $right$
Result: The original array now sorted by the **compareTo** within the bounds of $left$ and $right$ inclusive
if $left < right$ **then**
$mid = left + (right - left)/2$
$MergeSort(A, temp, left, mid)$
$MergeSort(A, temp, mid + 1, right)$
$merge(A, temp, low, mid, right)$
end

Algorithm 2: MergeSort psuedo code on ints

Function *merge(array A, array $temp$, int $left$, int mid, int $right$)*

Data: Arrays of **Comparable** objects A and $temp$, left and right endpoints, $left$ and $right$, and the middle index, mid

Result: The array A should be sorted within the bounds $left$ to $right$

for $k = left; k <= right; k++$ **do**
  $temp[k] = A[k]$
end
$lo = left$
$hi = mid + 1$
for $k = left; k <= right; k++$ **do**
  if $lo > mid$ **then**
    $A[k] = temp[hi++]$
  else if $hi > right$ **then**
    $A[k] = temp[lo++]$
  else if $temp[lo] <= temp[hi]$ **then**
    $A[k] = temp[lo++]$
  else
    $A[k] = temp[hi++]$
  end
end

Algorithm 3: merge method pseudo code on ints

8 Solution Description Part 3 - Big O Complexity

The final component of this homework is straightforward. You'll have several classes, depending on your thread picks, on algorithms, complexity, and runtime, but we want you to be conscious of it moving forward.

The idea of **Big-O** is that we want to model how long it takes to complete your algorithm given a problem of size n. In this case, we had you implement two algorithms - *InsertionSort* and *MergeSort*. The first was on the order of **O(N^2)** and the second $O(N \log N)$.

That distinction may not seem like much, but when it comes to datasets numbering in the hundreds, thousands, or even millions, this runtime complexity corresponds to the difference between seconds and hours.

We won't have you do any analysis, we just want you to appreciate the difference.

Before we move forward, you must have **InsertionSort** and **MergeSort** implemented correctly in **Sorting.java** or else this **Will Not Work**. In the given elements of this assignment, there's a class `SortingRuntime.java`. All you have to do is run the program with the following command

java SortingRuntime

The program will ask you what range of sizes you want to visualize: `Small`, `Medium`, or `Large`. The three ranges vary by an order of magnitude each time. What should appear is a graph plotting

the runtime on various input sizes for your two sorting implementations as well as java's built in sorting for comparison.

You should be aware that if you choose `Large`, it may take a while for the program to execute due to the effects of $O(N^2)$ asymptotic of Insertion sort. And by a while, I mean potentially 15 minutes. Do not use it as a metric for how good your implementation is. We primarily included it so you can see just how important asymptotics are, especially if you consider the size of some other datasets. For instance, Facebook has over a billion users and Google has over 30 trillion entries in its knowledge graph that supports search. `Small` and `Medium` will be much more revealing as to if your implementation follows the correct Big-Oh.

You should see three plots: two seemingly linear plots (this will be *MergeSort* and java's implementation of *MergeSort*) and one parabolic plot (*InsertionSort*). If you implemented *MergeSort* correctly, it should have a similar plot to java's implementation (it will be called `TimSort`).

NOTE: We will not be grading you on this graph. This is purely for your own visualization. Tips for testing your code are listed below.

9 Testing

Good testing is almost as important as good design. Everything works great in theory, but when it comes down to implementation, off-by-one errors and flipped signs can lead you down an hour long rabbit hole lamenting why your code isn't doing what you *know* it should be. Mistakes are unavoidable, but good testing can help mitigate the headache that they will cause.

The most helpful testing advice, as simple as it may seem, came from my senior year AI teacher. He stressed the concept of **CABTAB**: **C**ode **A** **B**it, **T**est **A** **B**it.

What does CABTAB mean exactly? In general, you should be designing your classes and implementing functionality in such a way that you have compartmentalized methods which independently accomplish some goal. These places are a great place to start testing. That being said, I don't mean to say that you should test every accessor, mutator, or constructor you define - that would be overkill. Instead, you should focus on higher complexity methods with lots of logic or arithmetic: boolean and numerical operations, low-level as they may be, are the bane of many programmers existence.

How does that apply for this `SiliconValley`? The main point of this homework is the implementation of `compareTo()` and the two sorting algorithms - and they will almost certainly be the root of most bugs.

What can we do then? Embracing **CABTAB**, we can start small - one `compareTo()` method at a time. After reading that, you may not be happy, but remember, five minutes of testing can erase thirty minutes of debugging. Back to `compareTo()`. In `TenXer.java` and all of its subclasses, we're comparing two objects based off a set of keys that have varying precedence (think primary, secondary, and tertiary keys). What you might want to do is create a couple instances of each subclass that are equal in all ways but one, then make sure that your `compareTo()` returns

the appropriate ordering.

Below is some sample code that I wrote to test my implementation, followed by its associated output. I used the `printSortPrint()` method in `SiliconValley` to simplify the printing. You can check out its javadoc to see what exactly it does.

```
TenXer[] sample;
// 1 - Cfos
System.out.println("\n1. Two Cfos");
TenXer gunther = new Cfo("gunther", 120000, 5, 50);
TenXer marcel = new Cfo("marcel", 120000, 5, 90);
sample = new TenXer[]{gunther, marcel};
SiliconValley.printSortPrint(sample);
// 2 - Ceos
System.out.println("\n2. Two Ceos");
TenXer pheobe = new Ceo("Pheobe", 500000, 12, 3);
TenXer joey = new Ceo("Joey", 500000, 8, 3);
sample = new TenXer[]{pheobe, joey};
SiliconValley.printSortPrint(sample);
// 3 - Software Engineers
System.out.println("\n3. Two SoftwareEngineers");
TenXer ross = new SoftwareEngineer("Ross", 80000, 10, 15, false);
TenXer rachel = new SoftwareEngineer("Rachel", 100000, 7, 15, true);
sample = new TenXer[]{ross, rachel};
SiliconValley.printSortPrint(sample);
// 4 - Cfo and Ceo
System.out.println("\n4. A Cfo and Ceo");
TenXer chandler = new Cfo("Chandler", 65000, 12, 50);
TenXer monica = new Ceo("Monica", 80000, 16, 2);
sample = new TenXer[]{chandler, monica};
SiliconValley.printSortPrint(sample);
// 5 - TenXer[]
System.out.println("\n5. Array of randomly generated TenXers");
sample = createTenXers(8);
Sorting.mergeSort(sample);
for (TenXer t : sample) {
   System.out.println(t);
}
```

And the associated output:

```
1. Two Cfos
Before Sorting:
gunther marcel
After Sorting:
gunther marcel

2. Two Ceos
Before Sorting:
Pheobe Joey
After Sorting:
Joey Pheobe

3. Two SoftwareEngineers
Before Sorting:
Ross Rachel
After Sorting:
Ross Rachel

4. A Cfo and Ceo
Before Sorting:
Chandler Monica
After Sorting:
Chandler Monica

5. Array of randomly generated TenXers
Albert  (Ceo)
      Salary:           106099
      Years Experience:     0
      Number of Startups:   0
John    (Ceo)
      Salary:           111029
      Years Experience:     9
      Number of Startups:   0
Kevin   (Ceo)
      Salary:            96534
      Years Experience:     0
      Number of Startups:   2
Louise  (Ceo)
      Salary:           104025
      Years Experience:    13
      Number of Startups:   3
John    (Cfo)
      Salary:           109984
      Years Experience:    16
      Money Skills:         2
Kevin   (Cfo)
      Salary:           113361
      Years Experience:    13
      Money Skills:         6
Jessica (SoftwareEngineer)
      Salary:           111722
      Years Experience:    14
      Github Stars:        29
      Knows Java:        No :(
Albert  (SoftwareEngineer)
      Salary:           108019
      Years Experience:    10
      Github Stars:        39
      Knows Java:         Yes
```

Hopefully this section gave you an idea on how to test your implementations for this assignment, and many more to come.

10 Checkstyle

Checkstyle counts for this homework. You may be deducted up to 20 points for having Checkstyle errors in your assignment. Each error found by Checkstyle is worth one point. Again, the full style guide for this course that you must adhere to can be found by clicking **here**.

One note, for this homework you will likely get checkstyle errors for an unimplemented `hashCode()` method. This happened in a previous homework and again, **we will not count off for those errors**.

This checkstyle error occurs because, in general, `hashCode()` and `equals()` represent the same idea - `hashCode()` maps your objects to integers where if two instances are equal by `equals()`, they should return the same `hashCode()`.

If you want to get rid of the errors anyway, you can implement `hashCode()` as follows:

```
public int hashCode() {
   return 0;
}
```

Come to us in office hours or post on Piazza if you have specific questions about what Checkstyle is looking for and how to fix Checkstyle errors.

First, make sure you download the `checkstyle-6.2.2.jar` from the link above. Then, make sure you put this file in the same directory (folder) as the `.java` files you want to run Checkstyle on. Finally, to run Checkstyle, type the first line into your terminal while in the directory of your Java files and press enter.

```
$ java -jar checkstyle-6.2.2.jar MyJavaFile.java
Audit done. Errors (potential points off):
0
```

11 Javadocs

For this assignment you will be commenting your code with Javadocs. Javadocs are a clean and useful way to document your code's functionality. For more information on what Javadocs are and why they are awesome, the online documentation for them is very detailed and helpful.

The relevant tags that you need to have are `@author`, `@version`, `@param`, and `@return`.
Here is an example of a properly Javadoc'd class:

```
import java.util.Scanner;
/**
* This class represents a Dog object.
* @author George P. Burdell
* @version 13.31
*/
public class Dog {
/**
* Creates an awesome dog (NOT a dawg!)
*/
public Dog(){
```

```
...
}
/**
* This method takes in two ints and returns their sum
* @param a first number
* @param b second number
* @return sum of a and b
*/
public int add(int a, int b){
...
}
}
```

Take note of a few things:

1. Javadocs are begun with `/**` and ended with `*/`.
2. Every class you write must be Javadoc'd and the `@author` and `@version` tag included. The comments for a class start with a brief description of the role of the class in your program.
3. Every non-private method you write must be Javadoc'd and the `@param` tag included for every method parameter. The format for an `@param` tag is `@param <name of parameter as written in method header> <description of parameter>`. If the method has a non-void return type, include the `@return` tag which should have a simple description of what the method returns, semantically.

ubsectionJavadoc and Checkstyle You can use the Checkstyle jar mentioned in the following section to test your javadocs for completeness. Simply add -j to the checkstyle command, like this:

```
$ java -jar checkstyle-6.2.1.jar -j *.java
Audit done. Errors (potential points off):
0
```

12 Turn-in Procedure

Submit your `TenXer.java, SoftwareEngineer.java, Cfo.java, Ceo.java` and `Sorting.java` file on T-Square as an attachment. When you're ready, double- check that you have submitted and not just saved a draft.

Keep in mind, unless you are told otherwise, non-compiling submissions on ALL homeworks will be an automatic 0. You may not be warned of this in the future, and so you should consider this as your one and only formal warning. This policy applies to (but is not limited to) the following:

1. Forgetting to submit a file
2. One file out of many not compiling, thus causing the entire project not to compile
3. One single missing semi-colon you accidentally removed when fixing Checkstyle stuff
4. Files that compile in an IDE but not in the command line
5. Typos

6. etc...

We know it's a little heavy-handed, but we do this to keep everyone on an even playing field and keep our grading process going smoothly. So please please make sure your code compiles before submitting. We'd hate to see all your hard work go to waste!

13 Verify the Success of Your Submission to T-Square

Practice safe submission! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.

2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.

3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.

4. Recompile and test those exact files.

5. This helps guard against a few things.

   (a) It helps insure that you turn in the correct files.

   (b) It helps you realize if you omit a file or files.[1] (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)

   (c) Helps find last minute causes of files not compiling and/or running.

[1] Missing files will not be given any credit, and non-compiling homework solutions will receive zero points. Also recall that late homework will not be accepted regardless of excuse. Treat the due date with respect. The real due date is 8PM Thursday. Do not wait until the last minute!